European Funds Knowledge Education Development
Wrocław University of Science and Technology
European Union European Social Fund

ZPR PWr – Zintegrowany Program Rozwoju Politechniki Wrocławskiej

Laboratory – List 9

Introduction

Disjoint set data structure is used when we have *n*- elements of a set and we need following operations:

- Creation of one-element sets for all *n* elements, which will be done on the begin – `makeSet(x)`. It means that we obtain a multiset, a set of one-element sets. We will never (during using this data structure) use this operation more times.
- Recognise if two elements belong to the same set. It is done by a function which will return the **representant** of the set to which belongs the argument – `findSet(x)`. It is assumed that if it is not called any other operation (in real it can be only union operation) between two calls for the function with the same argument, the returned answer has to be the same. Because this operation will be done the most often, an implementation has to minimize time of this operation.
- Make a union of two sets, exactly an union of sets S_x and S_y determined by two elements x and y which are the parameters of the operation `union(x,y)`. After this operation we assume that in the multiset there are no S_x and S_y, but it is now the sum of this two sets.

There are two possible inner representation of disjoint set data structure. Because it is known before start the constant `n`, both representations have to be implemented on an array.

One of the solution bases on one-way linked list implemented on an array. The elements are simple integer numbers which are also indexes of the array. In such a case the access to the element is done in constant time. In the array every element will store information about index of the next element (or -1 if there are no such an element), about the index of the head (which is also the representant), the size of the list (but only for the head this value is updated) and the index of the tail element (also updated only in the head). The size is used during union operation to connect a smaller list to a longer list (because in the first list we have to update the reference to the head of the second list) and the index of tail element to not spend time for searching the tail.

	0	1	2	3	4	5	6
Head	4	1	2	2	4	5	2
Next	-1	-1	3	6	0	-1	-1
Size	1	1	3	1	2	1	1
Tail	0	1	6	3	0	5	3

On above example we have 4 lists with elements (in order from a head to a tail):

- 2,3,6
- 4,0
- 1
- 5

Another implementation is with use of disjoint set forest. Like in previous one, we start from an array of elements. In one cell it is stored only two data: index of the parent element and a rank. In graphical representation it can be viewed as a set of trees. In this type of tree every node has only information where is the parent, and a root has itself as the parent.

But in this implementation during `FindSet` operation it has to be done a path compression. The representant is the root of the tree, so the procedure to find it goes from an element to its parent, than to the parent of previous parent and so on, till an node which has itself as a parent. This can be done in a recursive way. During come backing from recursion, the nodes on the path to a root update the field with parent with the root value. So, the next call of `findSet` for many argument will need many less steps to find the root (exactly 2 steps).

The union operation simple connect one representant to another reprentant as a parent. Which of the representant will be a root of a new tree depend on the rank. The node with smaller rank is connected to the one with bigger rank. If two considered ranks are equal, after connection we have to increase by one the rank of the root.

Remark

We will use following interface for data structures for disjoint sets:

```
public interface DisjointSetDataStructure{
	void makeSet(int item);
	int findSet(int item);
	boolean union(int itemA, int itemB);
}
```

List of tasks.

1. Implement class `DisjointSetLinkedList` which implements the above interface. The inner implementation has to use the idea presented above (and more details also in the lecture about implementing it on linked list, but stored in an array of element). Beside the interface, you have to implement the `toString()` method. The expected format of returned string is presented in an example in the appendix.
2. Implement class `DisjointSetForest` in a similar way like in the task 1. Because in this case there will be a set of trees in inner representation, so there is also a change of the format for the `toSting()` method.
3. There are prepared templates for classes (or interface) `DisjointSetDataStructure`, `DisjointSetLinkedList`, `DisjointSetForest`, `Main` in separated files.

For 100 points present solutions for this list till Week 11.
For 80 points present solutions for this list till Week 12.
For 50 points present solutions for this list till Week 13.
After Week 13 the list is closed.

There is a next page…

Appendix

The solution will be automated tested with tests from console of presented below format. The test assumes, that there are up to X different data structures for disjoint sets, which there are created as the first operation in the test. Each data structure can be constructed separately.

If a line is empty or starts from `'#'` sign, the line have to be ignored.
In any other case, your program should print an exclamation mark and write (copy) introduced a line and then, depending on the command follow the correct procedure / function.

If a line has a format:
`go <nMax>`
your program has to create an array of *nMax* Data structures for disjoint sets (without creation the data structure). The lists are numbered from 0 like an array of this data structures. Default current position is the number 0.

If a line has a format:
`linkedlist <size>`
`ll <size>`
your program has to call a constructor of a `DisjointSetLinkedList` with parameter `size` for the current position. If there is a data structure on the current position it will be overwritten. After creation call `makeset()` function for all items from 0 to `size`-1.

If a line has a format:
`dsf <size>`
your program has to call a constructor of a `DisjointSetForest` with parameter `size` for the current position. If there is a data structure on the current position it will be overwritten. After creation call `makeset()` function for all items from 0 to `size`-1.

If a line has a format:
`ch <`*`n`*`>`
your program has to choose a position of a number `n`, and all next functions will operate on this position in the array. There is *0<=n<nMax*.

If a line has a format:
`findset <item>`
`fs <item>`
your program has to call `findSet(item)` for the current data structure. The result of the search has to be written in one line.

If a line has a format:
`union <itemA> <itemB>`
your program has to call `union(itemA,itemB)` for the current data structure. The result of the unification has to be written in one line.

If a line has a format:
`show`
your program has to write on the screen the result of calling `toString()` for the current data structure. In a case of `DisjointSetLinkedList` there have to be a first line with a text "`Disjoint sets as linked list:` " and under this line it have to be every

linked list as a sequence of number from a head to a tail, separated by comma and one space. Lists have to be ordered depending of the value of the first element.
For `DisjointSetForest` the first line has to be equal "`Disjoint sets as forest:`" and every next will present the connection from every item to its parent in the format:
"`x -> y`", where `x` is an item and `y` is its parent. Values of `x` have to be from 0 do `size` - 1.

If a line has a format:
`ha`
your program has to end the execution, writing as the last line "END OF EXECUTION". Every test ends with this line.

An example test for this task:
INPUT:
```
#Test for Lab9
go 10
ll 8
findset 4
union 1 2
union 3 4
union 0 1
union 0 4
union 5 7
show
findset 1
findset 4
findset 7
union 2 3
ch 1
dsf 8
findset 4
union 1 2
union 3 4
union 0 1
union 0 4
union 5 7
show
findset 1
findset 4
findset 7
show
union 2 3
ha
```

OUTPUT:
```
START
!go 10
!ll 8
!findset 4
4
!union 1 2
true
!union 3 4
true
!union 0 1
true
!union 0 4
true
```

```
!union 5 7
true
!show
Disjoint sets as linked list:
1, 2, 0, 3, 4
5, 7
6
!findset 1
1
!findset 4
1
!findset 7
5
!union 2 3
false
!ch 1
!dsf 8
!findset 4
4
!union 1 2
true
!union 3 4
true
!union 0 1
true
!union 0 4
true
!union 5 7
true
!show
Disjoint sets as forest:
0 -> 2
1 -> 2
2 -> 4
3 -> 4
4 -> 4
5 -> 7
6 -> 6
7 -> 7
!findset 1
4
!findset 4
4
!findset 7
7
!show
Disjoint sets as forest:
0 -> 2
1 -> 4
2 -> 4
3 -> 4
4 -> 4
5 -> 7
6 -> 6
7 -> 7
!union 2 3
false
!ha
END OF EXECUTION
```